Final Report

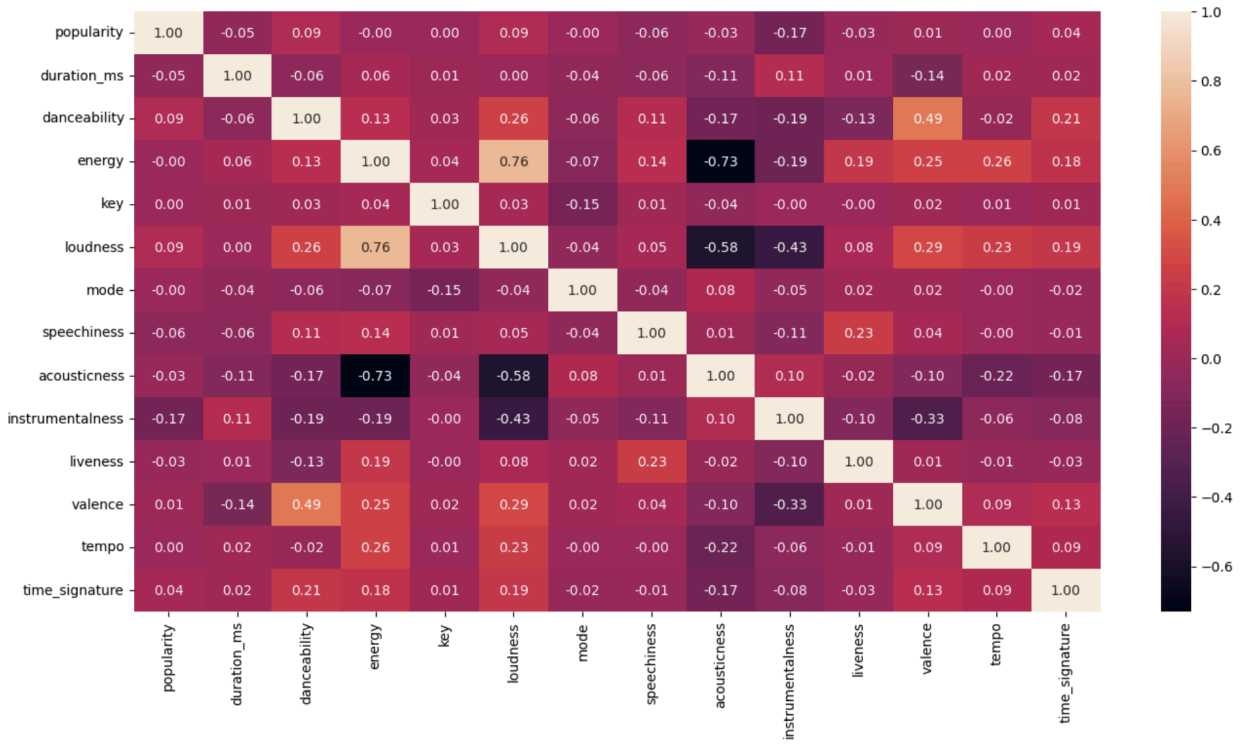
For my project, I decided to make a Spotify song/playlist recommendation system. Currently, I believe Spotify's existing recommendation system is not the best as, one, there is no explicit way of generating playlists, and two, the recommended songs always end up either getting repeated or are songs that are already in your playlists. That is why I tried to create my own recommendation system to, one, understand the workings behind a recommendation system, and two, see how my algorithm would compare to Spotify's algorithm. This project required 6 steps: obtaining the dataset, cleaning it, exploratory data analysis, preprocessing, coming up with the algorithm, and finally testing the algorithm. This project utilizes data management, machine learning, and exploratory analysis techniques, bridging core concepts from our lectures, such as data analysis, data cleaning, feature engineering, database exploration, and dataframe manipulation.

This project is important because it provides insight into how recommendation algorithms work, not just for Spotify songs and music but for many other things we encounter daily, such as movies/tv shows, social media, online shopping, and much more. Although this is not entirely a novel idea and many companies already spend a lot of time and resources into developing a robust recommendation algorithm, I was still excited to try and make my own because I was always curious about the inner workings behind these algorithms. I specifically chose to make a Spotify recommendation algorithm because one, just like many other people, I am an avid music listener, and two, I had worked on a Spotify-related project before so I was already familiar with some things such as the Spotify API and the information it returns regarding a song. Although I didn't use the Spotify API outright, the knowledge of it that I had before starting this project helped me understand my dataset faster so that I could focus on the more important aspects of the project such as the analysis of the data and the ML algorithms. I believe one issue in the current state of song recommendation algorithms is that many rely too much on collaborative filtering, which is the process of recommending things based on users with similar interests. The problem with this is that this tends to create clusters of users where they are exposed to only things that other users like, preventing them from branching out and exploring different interests. More specifically regarding Spotify, this is a problem for new and upcoming artists who want to be discovered but have difficulty doing so because their songs don't get recommended nearly as much as songs from more popular artists. That is why for this project, along with other reasons, I decided to create an algorithm that solely used content-based filtering, to see what recommendations would be like when only taking into consideration the features of a song in order to find songs with similar features.

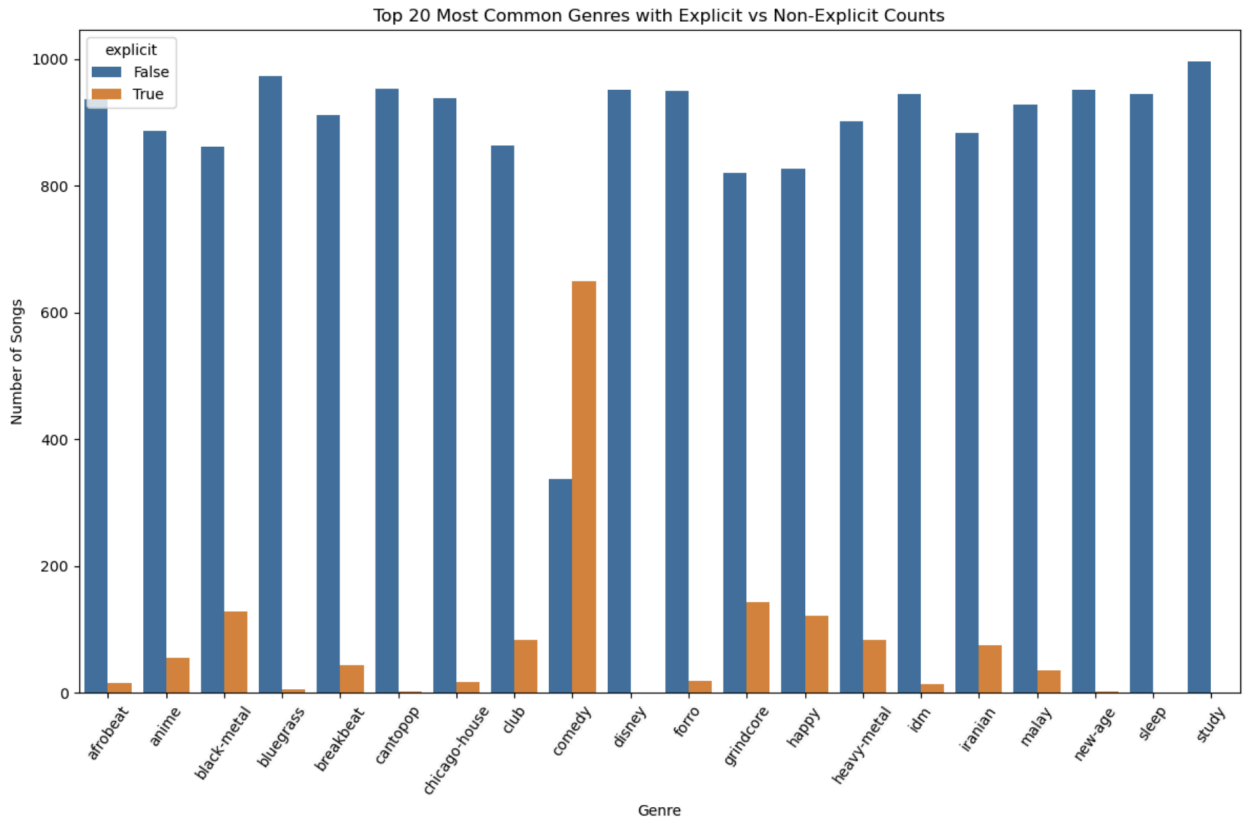
This project first started off with looking for a viable dataset. Even though I had prior knowledge of the Spotify developers API, I knew I wouldn't be able to create my own dataset using the API because of rate limits, so my next option was to find a public dataset. I used both Kaggle and Hugging Face to search for a good dataset and narrowed it down to 5 different datasets to choose from. I first started using a dataset from Kaggle with 30000 songs but during my process of cleaning the data, I found out that there were almost 5000 duplicate songs. That is when I decided not to use this dataset and look for a better one. I found three good datasets on Hugging Face, one with 1 million rows along with lyrics data but the problem with this one was that there was no data regarding the genre, which is believe was a big missing factor, especially considering I was performing content-based filtering, which relies solely on the features of a song, and genre plays a big part in that. Among the other two datasets, one had solely lyrics data and the other one was very similar to the Kaggle dataset with 30000 songs but this time with close to 115000 songs, meaning that even after removing the duplicates (which this dataset had 25000), there were still over 75000 unique songs in the dataset. I ended up using this dataset and I wanted to merge it with the dataset with only lyrics but the problem was that there were no unique identifiers between the two datasets that would allow for merging without losing any songs. Also, the lyrics dataset only had 55000 songs without accounting for duplicates so I knew that even after attempting to merge the data as much as possible, I would be left with very few songs to choose from. I wanted the lyrics data because in my project proposal, I had wanted to create a recommendation system where a sentence is inputted and songs are recommended based on that input, and this can only be done when there is lyrics data present in the dataset. Since I couldn't obtain the lyrics data, I unfortunately could not complete this task that I had outlined in my proposal. However, in the end, I was still able to get a good dataset, with a lot of songs, many features, and very little missing data, so that I could accomplish my main goal of creating a content-based recommendation system.

After obtaining the dataset, my next step was to clean the data. This part was quite simple because there was not that much missing data (only 0.0009%) so I just removed them. There was a lot of duplicate data (around 20%) but there is not much you can do about that other than to simply remove it so that's what I did. One issue I encountered was when I removed the rows with duplicate track ids but after I finished my recommendation system, I was still getting duplicate songs by the same artist. In order to fix this, I went back and looked for rows that had duplicate song names and artists and there were around 8000, so I had to also remove these rows.

Now that the dataset was ready, I began to perform data analysis on the dataset. The first thing I did was look at the columns and their respective data types along with statistics regarding the numerical columns (mean, std, min, etc.). Then, I graphed some plots using seaborn and matplotlib in order to visualize the overall data better. One graph was the heatmap of the correlations between the numerical variables, which showed that loudness & energy had the largest positive correlation while acousticness & energy had the largest negative correlation. In order to visualize these correlations some more, I graphed a pairplot between these three variables, which showed that there was in fact a pattern between these variables.



The above visual is a heatmap of the correlations between different numerical variables in the dataset



The above visual is a barplot of the 20 most common genres along with their explicit & non-explicit song counts

Finally, I graphed a barplot (as seen above) of the 20 most common genres and their distribution between number of non-explicit songs versus number of explicit songs. Looking at the graph, it showed that genres such as ‘disney’, ‘sleep’, and ‘study’ had little to no explicit songs (which is expected), while genres such as ‘black-metal’, ‘grindcore’, and ‘heavy-metal’ have a large amount of explicit songs. However, the number of explicit songs was still significantly less than the number of non-explicit songs for almost all of these genres except one, which was ‘comedy’, where there were almost twice as many explicit songs compared to non-explicit songs. One reason for this could be the fact that this dataset doesn’t only include songs but maybe also podcasts and other forms of media on Spotify, which may contain significantly more explicit words compared to songs.

After the data visualization, I explored the data some more by performing different SQL queries. First, I loaded the songs dataframe into an SQL table and then checked whether the df was inputted successfully. After that, I ran some queries, the first one being the top 20 genres ranked by popularity. I saw some obvious answers such as k-pop, pop, and hip-hop, along with others like EDM and house. K-pop was the most popular with an average popularity rating of 59.3. However, the only genre on this popularity list that was also in the 20 most common genres was ‘anime’, meaning that the rest could have a high popularity due to the lower number of songs it has. This was further proved when I ran another SQL query where I obtained the 10 most popular songs out of the 30 least common genres. Out of the 10, the first 8 were in the top 20 most popular genres, once again showing that genres with higher average popularity tend to have lower song counts. My next two SQL queries involved obtained the average feature scores for the categorical variables ‘explicit’ and ‘key’. For the ‘explicit’ variable, I found out that explicit songs tend to have higher popularity, danceability, and energy ratings than non-explicit songs while non-explicit songs tend to be longer on average. For the different keys, no key really stood out because they all had feature values that were close to the overall average, except for key 3 (D#/E ♭), where the average tempo was 118 bpm, differing from the others which were around 122 bpm. The next SQL query didn’t really provide much insight into the data as a whole but I just ran it to find out which songs were the most popular in each genre. As expected, the most popular genres had top songs with high popularities while the less popular genres had less popular top songs. My final SQL query was to find “hidden gem” songs, which once again do not provide much information regarding the overall data but it was still interesting to see. I found these songs by going by genre and selecting a song where the popularity was lower than average but its other features were quite high. While knowing these hidden gems might not be important, knowing of their presence is, because it shows that there are songs out there that might not be recommended to others because of their low popularity score, when in reality, that should not be the case. Having a low popularity doesn’t mean that a song should not be recommended because there may be people out there that still would like to listen to it, ultimately raising its popularity. This is where I believe a flaw is present, because if a song’s popularity goes down, it may never come back up because it won’t get recommended as much anymore, causing its popularity to stay low and not get a chance to go back up again because less people get to listen to it. This is

where I wonder where the popularity metric comes from because this score and many of the other scores (such as danceability, valence, & energy), are all somewhat subjective even if claimed not to be, and ultimately, the algorithms from Spotify determine these scores, which could be flawed. Therefore, in a way, Spotify's algorithms determine the success of an artist/song, which is interesting to think about since these algorithms are developed by people who have their own opinions, meaning that these people essentially play a role in the popularity of a song. While this could be seen as a problem, there is not really a fix to it because in the end, a recommendation algorithm will always be biased in some way or another, no matter how much effort is put into making it unbiased, because the nature of humanity is in and of itself biased. The only way to remedy this issue is to mitigate the biases as much as possible, which I am sure Spotify is constantly trying to do.

After the exploratory data analysis and understanding of the dataset, it was time to move onto the machine learning portion of the project where I made the actual algorithm. However, before making the algorithm, some preparation/preprocessing was required, and this was in the form of one-hot-encoding (OHE) and MinMax scaling. First, I performed OHE on the categorical variables such as genre, key, mode, etc. I chose to one-hot-encode instead of label encode because label encoding is usually for categorical variables where order matters. However, for all categorical variables in this dataset, order did not matter and therefore, I didn't want there to be an unnecessary weightage put on certain categories over others, so one-hot-encoding is the way to prevent this from occurring. Next, I needed to normalize the columns where the values were not between 0 and 1, which were only a few columns (popularity, duration, loudness, and tempo), as the rest were already between 0 and 1. I chose to normalize these values by using MinMax scaler rather than Standard scaler because MinMax scaler is better for preserving the relationship between values which is important for algorithms in which the distance between the values matters, such as the k-nearest neighbors (KNN) algorithm, one that I will be using. After normalization, I then split up the data frame into a features dataframe, which held only the numerical values, and a names dataframe, which held the titles, artists, and album names of each song. The features dataframe is very important because it holds the multi-dimensional feature vectors of each song, which is essentially the backbone of a machine learning algorithm. Finally, I made sure each data frame had the track id of each song so that it could be used as an identifier between the two dataframes.

After preprocessing, I was ready for the most important part of the project, which was writing the actual recommendation algorithm. First before starting the ML portion of the recommendation algorithm, I wrote some helper functions that would help speed up the testing of the algorithm. These algorithms consisted of obtaining a random song or playlist from the dataset and then two more functions that obtained the single feature vector (the feature vector is the row of the song from features dataframe) of a song based on the inputted track id, or the aggregated playlist vector (mean of all song vectors) based on the inputted array of track ids that represented the playlist. Not only would these two helper functions return the song or playlist feature vector, but it would also return the features data frame without that song or playlist songs

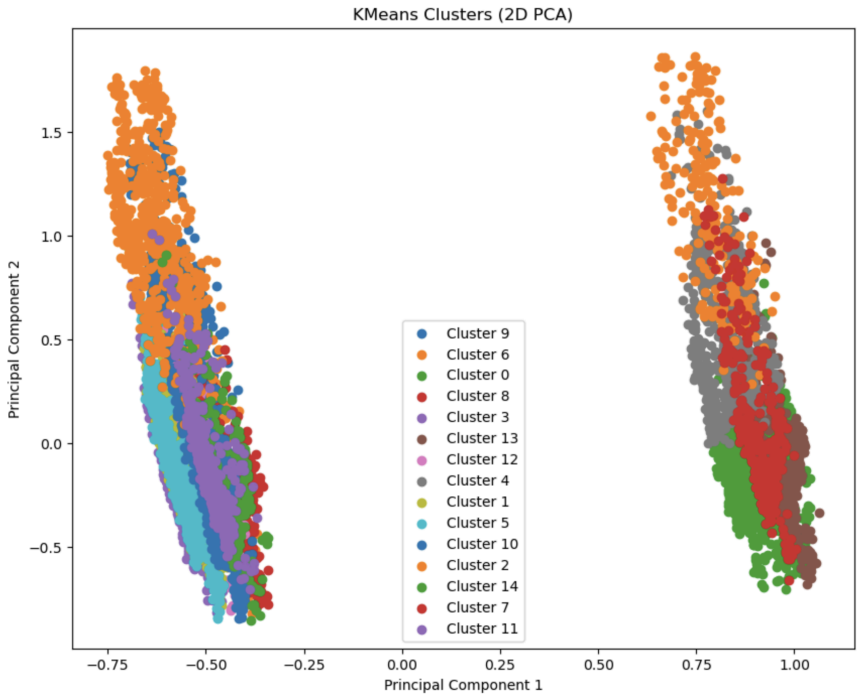
in it, so that when the recommendation algorithm uses this new feature dataframe, the recommended songs will not contain any of the input songs. Finally, my last helper function was created in order to help visualize the recommended songs since it required taking the recommended track ids from the features dataframe and finding the corresponding song titles/artists/album from the song_names dataframe.

After writing the helper functions, it was time to write the ML functions. I mainly used the Scikit-learn Python library for these functions and the first function I wrote was a k-Nearest-Neighbor algorithm which found the closest neighbors to the input feature vector based on cosine-similarity, which is the angle between these high-dimensional vectors. I tested other k-NN methods such as Euclidean distance and Manhattan distance but the only method that returned logical and relevant recommendations was cosine similarity. One example is when the input song was in a different language such as German or Japanese, and cosine similarity was the only k-NN method that returned recommended songs of the same language; the other two returned seemingly random songs. I also implemented a bit of randomness/variability into this function in that based on the playlist size, the function obtain the playlist size * 10 closes number of songs and then randomly sampled the correct playlist size number of songs from this larger pool (of closely related songs). Therefore, if the same song was inputted multiple times, different recommendations would be given back each time.

```
from sklearn.metrics.pairwise import cosine_similarity, euclidean_distances, manhattan_distances

# To supress warning
pd.options.mode.chained_assignment = None

# Find k-Nearest-Neighbors using cosine similarity
def get_similar_songs(feature_vector, features_without, playlist_size):
    feature_vector = feature_vector.values.reshape(1,-1)

    # after some testing, cosine_similarity seems to be the best for finding most similar songs
    # cosine similarity finds vectors with closest cosine angle

    features_without['similarity'] = cosine_similarity(features_without.drop('track_id', axis = 1).values, feature_vector)
    #features_without['similarity'] = euclidean_distances(features_without.drop('track_id', axis = 1).values, feature_vector)
    #features_without['similarity'] = manhattan_distances(features_without.drop('track_id', axis = 1).values, feature_vector)

    # obtain large pool of similar songs (10*playlist_size)
    top_songs = features_without.sort_values('similarity', ascending = False).head(playlist_size*10)

    # sample from this large pool (in order to create variability)
    sampled_top_songs = top_songs.sample(playlist_size)
    track_ids = sampled_top_songs['track_id'].values
    return track_ids
```

The above code is finding the k-nearest neighbors using cosine similarity

This cosine similarity recommendation algorithm was already quite good but in order to see if I could do better, I decided to test out another machine learning algorithm. This time, it was KMeans clustering, where the model would cluster the data into a set number of clusters (15 in my case) based on feature similarities and then only recommend songs that belong to the same cluster as the input song/playlist. First I trained this model and then assigned a 'cluster' column to the features dataset, which made the dataset divided into clusters. I then visualized these clusters using matplotlib and Principal Component Analysis (PCA), which reduces high-dimensional vectors down to two main components so that the high-dimensional vectors can be visualized in a 2D scatter plot. Finally, I created the function to obtain the recommended songs based on the cluster of the input song/playlist. At first, I randomly sampled from these clusters but because the clusters were still quite large, there were still some differences between the songs within the same clusters, which is why my recommendations did not seem to be great. That is why I decided to come up with a hybrid approach, where I combined KMeans clustering with cosine similarity so that I could recommend the most similar songs within the same cluster. This approach seemed to work better but at times, it was no different than my first algorithm that just used cosine similarity. That is why, in order to visualize the differences, I created my final function using matplotlib and PCA once again, which would visualize the similarities and differences between my two algorithms in terms of how close each recommended song (from either function) was to the input song/playlist.



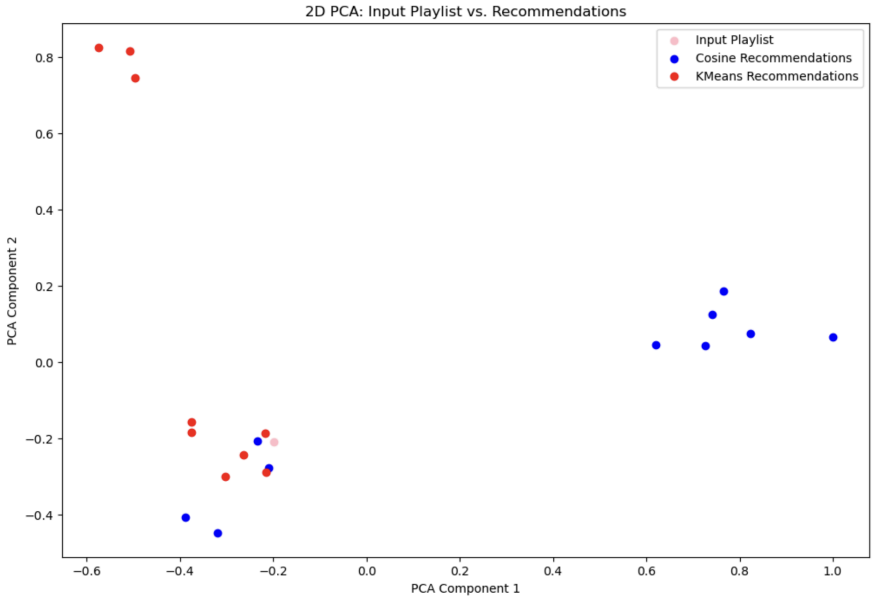

```
from sklearn.cluster import KMeans

# create 15 clusters
n_clusters = 15
kmeans = KMeans(n_clusters=n_clusters)
features_copy = features.copy()

# create 'cluster' column to associate each song feature vector with a certain cluster
features_copy['cluster'] = kmeans.fit_predict(features_copy.drop('track_id', axis = 1))
```

The above code is splitting up the features dataframe into 15 different clusters

```python
def get_cluster_recommendation(track_ids, df_features, n_recommendations=5):
    reco_ids = []
    if (len(track_ids) == 1): # input is a song
        # obtain song feature vector
        song_vector, features_without = get_song_vector(df_features.drop('cluster',axis=1), track_ids)

        # obtain cluster number based on track_id
        cluster_id = df_features[df_features['track_id'] == track_ids[0]]['cluster'].values[0]

        # filter features to only include the track's cluster
        cluster_songs = df_features[(df_features['cluster'] == cluster_id) & (df_features['track_id'] != track_ids[0])]

        # use cosine similarity to obtain similar songs within that particular cluster
        reco_ids = get_similar_songs(song_vector, cluster_songs.drop('cluster', axis=1), n_recommendations)
    else: # input is a playlist

        # obtain playlist feature vector
        playlist_vector, features_without = get_playlist_vector(df_features.drop('cluster',axis=1), track_ids)

        # turn playlist vector into df (needed for next line --> kmeans.predict)
        playlist_vector_df = pd.DataFrame(playlist_vector.values.reshape(1, -1), columns=playlist_vector.keys())

        # obtain cluster id of playlist vector by predicting it using the kmeans model we initially made
        cluster_id = kmeans.predict(playlist_vector_df)[0]

        # filter features to only include the playlist's cluster
        cluster_songs = df_features[(df_features['cluster'] == cluster_id) & (~df_features['track_id'].isin(track_ids))]

        # use cosine similarity to obtain similar songs within that particular cluster
        reco_ids = get_similar_songs(playlist_vector, cluster_songs.drop('cluster', axis=1), n_recommendations)

    return reco_ids
```

The above function combines K-means clustering and cosine similarity

The final step of this project was to finally test out my recommendation algorithms. Because of all the helper functions I created, this part was not too difficult. My first test was obtaining a playlist of 10 songs after inputting one song. First I obtained a random track id, then I inputted this track id to get the song feature vector. Then I got the recommended songs using cosine similarity and then displayed these songs. Using the same song id, I obtained another recommended playlist of 10 songs but this time using the KMeans function. I then used the function that utilized PCA to reduce the high-dimensional feature vectors down to 2 dimensions so that all of the recommended songs and the input song could be viewed as a 2D scatterplot. This way, I can visualize how close each function was at recommending songs that were similar to the input song. After one particular test, the graph showed that the KMeans recommendations had more closely related songs to the input song than cosine similarity recommendations. Obviously, this graph will look different each time new recommendations are made, which means that KMeans might not always have better results each time.



My next test included obtaining a 5 song playlist from a randomly generated 5 song playlist. This results for this test were not great because a completely random playlist can have vastly different songs, so recommendations might not be similar to any song in the playlist because both of my algorithms blend each song feature vector together into one aggregated playlist vector. This aggregated vector may not be similar to any song in the playlist if all of the songs are vastly different, and so the recommendations might not be relevant at all. Therefore, not being able to recommend songs using a very diverse playlist as an input is a big limitation that I acknowledge is present in my project. That is why for my last test, I used an already recommended playlist from a song as part of my input. Therefore, this initial recommended playlist will have songs that are already similar to each other so that when the recommendation is run again, the algorithm will return even more similar songs. After this test, my project was complete but I ran the tests a couple more times in order to ensure that everything was working properly.

After completing my project, there were some aspects outlined in my proposal that I could not fulfill in my final project. One was recommending songs based on temporal data. I wanted to be able to recommend songs based on the time, such as calmer songs during the night and upbeat songs in the morning, but unfortunately, I did not have access to any time data that would allow me to create this kind of recommendation system. Similarly, I wanted to create recommendations based on sentence prompts, but as mentioned at the start of my report, I did not have access to a good lyrics dataset that would allow for this. Finally, the last thing I couldn't fulfill was a collaborative-filtering algorithm because once again, I did not have access to any user data. I definitely could have simulated my own dataset but I was afraid the results would not make any sense if I made the dataset incorrectly. I had also wanted to test how good a recommendation algorithm could be by only using content-based filtering, which is why I decided not to continue with implementing collaborative filtering. Therefore, all of this shows how my greatest bottleneck in this project was access to data. That is why large companies like Spotify are able to achieve what they are doing right now; their accessibility to all the different kinds of unique and complex data not available to the public allows them to do things no one else can do unless they have access to the same data.

In terms of evaluating my algorithm for accuracy, there is not really much I can do because there is not really any value present in the data to test it against. That is why I think there is a big limitation when it comes to testing prediction algorithms; the only way you can truly test its accuracy is by getting real user feedback. However, while working on my project, some key findings I discovered were that cosine similarity is very good at recommending songs. However, this might not apply for other recommendation systems such as movies or social media algorithms. Another result I discovered is that the KMeans clustering combined with cosine similarity makes a slightly better model but it also introduces more randomness and variability, which can lead to worse results at times.

At the start of my report, I mentioned that recommendation algorithms should use collaborative filtering less.. I wanted to see how well content-based filtering algorithms compare

to collaborative filtering algorithms. In the end, I came to the conclusion that content-based filtering is still very strong on its own. Even though my algorithms were not that complex, I still was able to get decent results in terms of my recommendations. My algorithms are also probably advantageous in terms of speed and space efficiency, compared to the complex and robust algorithms developed by Spotify. However, my algorithm also has limitations because the efficiency comes at the cost of accuracy in terms of providing relevant results, since at times I was given recommendations that were not that relevant to the input song. This is where Spotify beats my algorithm, because they have much more data and resources to build a strong recommendation algorithm that doesn't falter and always provides relevant results.